ATARI® PROGRAM exchange

BASIC UTILITY FOR RENUMBERING PROGRAMS (BURP)

by

Gifford O. Kucsma

INSTRUCTIONS

USER-WRITTEN SOFTWARE FOR ATARI PERSONAL COMPUTER SYSTEMS

APX-10046

APX-20046

TRADEMARKS OF ATARI

The following are trademarks of Atari, Inc.

ATARI
ATARI 400 Home Computer
ATARI 800 Home Computer
ATARI 410 Program Recorder
ATARI 810 Disk Drive
ATARI 820 40-Column Printer
ATARI 822 Thermal Printer
ATARI 825 80-Column Printer
ATARI 830 Acoustic Modem
ATARI 850 Interface Module

BURP

BASIC Utility for Renumbering Programs

A BASIC Program Renumbering
Routine for the Atari Personal Computer System

by

Gifford O. Kucsma

BURP

Table of Contents

I. INTRODUCTION

What is it?

During the development of programs written in BASIC, it occasionally becomes desirable to renumber the lines of the program. This may be because the programmer finds the need to insert additional program lines, or simply to clean up the program so that the line numbers provide a more effective indication of the relative position of lines within the program.

Other BASIC renumbering programs have been offered to users of the Atari Personal Computer System, but have suffered from several failings or weaknesses. Routines have been published which renumber the lines, but not the references to the lines (GOTO's, GOSUB's, TRAP's, etc.) - a virtually pointless exercise. Other renumbering programs will do this, but require the availability of an Atari 810 Disk Drive, a significant economic consideration. In addition, these renumbering utilities fail to recognize the use of variables and arithmetic expressions as line number references.

BURP has none of these drawbacks. It offers a renumbering capability to those who must be content with an Atari 410 Program Recorder, although it additionally offers improved performance to owners of an Atari 810 Disk Drive. BURP is indifferent to the presence of cassettes or diskettes because it operates entirely in RAM. BURP renumbers references to lines as well as the lines themselves. It does not ignore the possibility and implications of variables and arithmetic expressions used as line number references.

How does it work?

You have a program resident in RAM, which you have typed in, LOADed, or ENTERed. You merge the renumbering routine with your program by ENTERing BURP.

You then execute the routine by keying in a direct mode GOTO to the BURP routine.

BURP queries you for the new starting line number and the value of the line numbering increment you wish to use.

BURP then proceeds to scan your tokenized RAM-resident BASIC program, line by line, looking for any references to line numbers. Any line numbers referred to are entered into a table, together with the RAM location of the reference.

On completion of this process, a second scan of your tokenized program is made, renumbering the lines. As each line is given a new number, the old number is looked up in the table. If it appears there, the new line number is POKEd into the corresponding RAM address of the referencing statement.

What are BURP's limitations?

There are essentially three limitations. The first is a "logical" limitation, for which there is literally no solution. If you write a program in such a way that references to line numbers are developed as the program is RUN, it is not possible to determine, by looking at a reference in the form of a variable or arithmetic expression, what it's value will be when the program is RUN. The author of BURP is of the opinion that the use of this technique is not a particularly good programming practice. Of course, you might say "well, sure...because he can't handle it!". Not so! If you have ever tried to decipher or debug a program which was written using this technique, you will understand and appreciate this opinion. However, the author has made provision for this practice.

Rather than make no attempt to accommodate variables and arithmetic expressions used as line number references, BURP was written to handle them in the same manner as the RESequence function of the best BASIC interpreters available on commercial time-sharing systems.

If you follow a simple rule in formatting arithmetic expressions as line number references, BURP will handle them properly:

> If the first element of the arithmetic expression is a numeric constant, the renumbering utility will treat that element as a line number, and attempt to renumber it.

For example, BURP will properly handle the following statements, assuming that 230 is a valid line number:

```
150 GOTO 230+A
160 GOSUB 230+(X*2)
170 IF R=S THEN GOTO 230/P
```

On the other hand, the following statements are some examples of references that cannot be renumbered:

```
100 GOTO A
110 GOTO X+Y
120 GOSUB W+230
```

However, in the case of lines 100 through 120 above, BURP will pause to warn you of the existence of these references, and give you the opportunity to cancel the renumbering process. You can then study your program to insure that not renumbering the reference is all right, or else modify the statement in question so that it can be properly handled.

For further information, read the "Troubleshooting and Cautions" section.

The second limitation of BURP is a "physical" one. The same thing that gives BURP its performance advantage may, in some cases, constitute a limitation. Because it co-exists in RAM with your BASIC program, it operates at RAM speeds. However, this means that you must have enough RAM to contain (a) your BASIC program, (b) BURP, and (c) a dimensioned array sufficient to contain an entry for each line number reference.

The third limitation is a relatively minor one. Because the renumbering routine is itself a BASIC program, it must use some line numbers. Obviously, the line numbers used by your program cannot duplicate those used by BURP. The BURP routine is distributed with line numbers in the range of 32500 through 32761. Consequently, your program cannot use a line number higher than 32499. If, in the process of renumbering your program, a new line number is developed which would violate this rule, BURP will terminate, advising you of the potential conflict. You can then begin the renumbering process again, perhaps using a different increment or lower starting line number value.

What are the minimum machine requirements of BURP?

The condensed, or "mashed" version of BURP occupies 3300 bytes of RAM. The remainder of the RAM requirement depends on the size of the program you wish to renumber, and the number of line references within it. Since the Atari Operating System itself will occupy 3058 bytes of your available RAM space, and 3058 + 3300 = 6358, the realistic minimum equipment requirement would be:

an Atari 400/800 with 16K RAM
an Atari BASIC Language Cartridge
an Atari 410 Program Recorder

Of course, either an Atari 410 Program Recorder or Atari 810 Disk Drive can be used for the process of LOADing, SAVEing, and ENTERing programs.

While it would be nice if BURP could operate in less RAM (some suggestions for further condensing it are provided in the Advanced Technical Information section), you shouldn't get excited over giving up 3300 bytes. Many programs to be renumbered will have this much additional RAM available for graphics workspace and arrays, which will be unoccupied until the program is actually RUN. To put things in perspective, DOS II grabs 5579 bytes, in addition to the 3058 occupied by the Operating System. Therefore, in a pinch, DOS II users could run BURP in cassette mode, with their diskette(s) temporarily detached.

Are any "tricks" used?

One of the author's objectives, aside from satisfying his own requirement for a renumbering utility, was to meet the challenge of performing this task without resorting to functions which would not be available to the average person. Although, using Assembler Language, much higher performance could have been achieved, BURP was written entirely in BASIC, without so much as a POKE at machine language and the USR function. As such, it provides not only a useful tool, but an interesting example of the flexibility of Atari BASIC.

II. GETTING STARTED

Available on both cassette and diskette, BURP is written in LIST (untokenized) form. The tape contains only the "mashed" version. The diskette contains a mashed version and an unmashed version. The unmashed version of BURP is provided primarily for your perusal of the program. The mashed version is the one that you will actually use. See Section VI of this manual for a listing of the unmashed program.

On receiving any piece of software, the first thing you want to do is protect yourself against coffee spills, cigarette ashes, and the variety of jeopardies that hover about your "data center". Make a backup copy. For cassette, the procedure is as follows:

> With your Atari BASIC cartridge inserted in the left cartridge slot, turn your computer on.
>
> Insert the BURP cassette in your Atari 410 Program Recorder in position to read Side 1.
>
> Press REWIND, to insure that you are at the beginning of the tape, then press PLAY.
>
> Type NEW and press RETURN once. This will ensure that RAM is free of any other program code.
>
> Type ENTER "C:" and press RETURN twice. This will read the BURP program into RAM.
>
> Insert a fresh cassette in your Atari 410 Program Recorder.
>
> Press REWIND, to insure that you are at the beginning of the tape. Then, press RECORD and PLAY, to put the unit into "record" mode.
>
> Type LIST "C:" and press RETURN twice. The BURP program will then be LISTed on your cassette in untokenized form.

For diskette, use DOS II menu selection C (if you have more than one disk drive) or O (if you have only one disk drive) to duplicate the files. The file names are BURP (mashed version) and BURP.REM (unmashed version). If you transfer the cassette program to diskette, be sure to store it on diskette in LIST format (e.g., LIST "D:BURP") so that you can ENTER it into RAM that is already occupied by the program to be renumbered.

III. USING BURP

With an Atari 410 Program Recorder

Place your BASIC program in RAM, by typing it in, LOADing or CLOADing it, or ENTERing it.

If you have typed it in, or made modifications to it, CSAVE, SAVE, or LIST it to protect it from loss should a problem be encountered in renumbering it.

If possible, RUN your program, to satisfy yourself that it is functioning properly prior to renumbering it.

Press SYSTEM RESET. This will ensure that you haven't left any non-standard values POKEd into the Operating System that might interfere with the successful execution of BURP.

Type CLR and press RETURN once. This will insure that, if your program has DIMensioned any arrays, that space will be released and thus be available for use by BURP.

Insert your BURP cassette in the Atari 410 Program Recorder, positioned to read Side 1.

Press REWIND to insure that you are at the beginning of the tape, then press PLAY to put the recorder in "read" mode.

Type ENTER "C:" and press RETURN twice. This will read BURP into RAM, co-resident with your BASIC program to be renumbered.

Type GOTO 32500 and press RETURN. This will begin execution of BURP.

BURP will now clear the screen and display:

```
New start line number?_
```

Respond with a numeric value indicating the line number that you would like your renumbered program to begin with, then press RETURN.

Using BURP

With an Atari 410 Program Recorder

(Continued....)

BURP will then query you:

```
Increment?_
```

Respond with a numeric value indicating the line number increment you would like to use, then press RETURN.

BURP will now commence scanning your BASIC program, line by line. As it does so, it will display:

```
Scanning line: nnnn
```

The number nnnn will give you an indication of the progress which BURP is making through your program lines.

If, during this phase, BURP encounters a line number reference in the form of a variable name, it will display the line of your program containing that reference, "honk" at you, and ask you if you would like to continue. For example:

```
350 IF A=B THEN GOTO XYZ
Variable reference.
Go ahead anyhow?_
```

If, upon examination of the line in question, you decide that it poses no problem that you can't cope with, respond with a "Y" followed by pressing RETURN. BURP will then ignore the reference, erase the line display from the screen, and continue to scan your program.

On the other hand, if you aren't sure what the consequences of ignoring the variable reference will be, respond with an "N" followed by pressing RETURN. BURP will then terminate, leaving your BASIC program untouched. Note that for safety's sake, any response other than "Y" will be considered equivalent to "N".

Using BURP

With an Atari 410 Program Recorder

(Continued.....)

During this scanning phase, BURP is doing a preliminary calculation of what the new line numbers are going to be. If it determines that renumbering your program will result in a line number greater than 32499, it will display:

> New line number exceeds the
> maximum allowable value.

There are no options when this occurs, and BURP will terminate its processing. You should then begin again, using a lower starting line number or smaller increment value.

It is during this phase that BURP is entering line number references in an array that is limited by the RAM space available. If it encounters more line number references than can be handled in the available space, it will display:

> Insufficient RAM to handle
> more than nn references.

BURP will terminate its processing. If you can either make more RAM available or reduce the number of line number references within your program, you may then begin again.

Once BURP has completed its scan of your BASIC program, it will sort the line number references to facilitate their rapid identification during the actual renumbering phase. When the sort phase has been entered, BURP will display:

> Sorting the references: nnnn *

As the sort progresses, the value nnnn will indicate an increasing count of the number of line number references which were detected. The asterisk will flash, indicating that sorting is taking place and reassuring you that, when a delay occurs, processing is in fact proceeding

Using BURP

With an Atari 410 Program Recorder

(Continued.....)

Upon completion of the sort phase, BURP will display:

Renumbering bnnn as annn

The two values bnnn and annn will indicate the before and after line numbers, respectively. Once this display appears, BURP has commenced the actual renumbering process and any premature termination of BURP will leave you with a damaged program, as it will be only partly renumbered.

Upon successful completion, BURP will generate a tone signal, READY will appear on the screen, and the cursor will be restored.

At this point your BASIC program should be visible in RAM with the new line numbers applied. You may want to RUN it, to make certain that any variable line number references haven't caused a problem.

The final step is to make a copy of your renumbered program. Insert a fresh cassette in your Atari 410 Program Recorder.

Press REWIND, then press RECORD and PLAY, to place the recorder in "record" mode.

Type LIST "C:",0,32499 and then press RETURN twice. Your renumbered BASIC program will then be LISTed on the cassette. LISTing your program this way is necessary to strip it of the BURP program, which would otherwise be tacked on the end of it. If you don't mind this (you may want to keep BURP appended to a program that is under development), you can just as well CSAVE all of RAM or do a LIST "C:" without limiting it to the range 0 through 32499.

Using BURP

With an Atari 810 Disk Drive

The procedure for using BURP from an Atari 810 Disk Drive is essentially the same as that for the cassette unit.

The principal difference is that any of the following commands would be replaced as follows:

LIST "C:"	use	LIST "Dn:filnam.ext"
LIST "C:",0,32499	use	LIST "Dn:filnam.ext",0,32499
CLOAD	use	LOAD "Dn:filnam.ext"
LOAD "C:"	use	LOAD "Dn:filnam.ext"
ENTER "C:"	use	ENTER "Dn:filnam.ext"

The file name of the mashed version is BURP; the file name of the unmashed version is BURP.REM.

If you have an Atari 810 Disk Drive, using BURP will be much more pleasant since the operations required to ENTER it into RAM and LIST your renumbered program back onto diskette will be much quicker.

However, as noted elsewhere, if you find that you need more RAM space to successfully renumber a program, detach your Atari 810 Disk Drive(s) and reinitialize your system to run without the Disk Operating System resident in RAM.

IV. TROUBLESHOOTING AND CAUTIONS

There isn't much potential for disaster using BURP, providing you take some of the basic precautions that you would observe any time you were working on a program.

Because BURP operates in RAM, it doesn't touch the previous version of your program. Therefore, even if you make a mess of it by BREAKing into BURP while it's in the final renumbering phase, you still have your virgin copy out on tape or diskette. However, just to be nice, once BURP has displayed the message "Renumber nnnn as nnnn" on the screen, please don't interrupt it. If you do, you will have BASIC statements in RAM, some of which have been renumbered and others which haven't.

In the introduction, I discussed the use of variables as line number references. This is acceptable, but it's up to you to ensure that the line numbers which are derived from those variables when you RUN your program are the right ones. For example, you might have a sequence like this:

```
150 LET X=TRNSTYPE
160 GOTO 200+X
```

Now, you may have written your program on the assumption that X will always have a value of 1, 2, or 3. There may not actually be a line 200, but you do have lines 201, 202, and 203. One thing that BURP, or any other utility, can't do, is to renumber a reference to a nonexistent line number. Therefore, if you run these BASIC statements through BURP, you're going to have to handle the change to line 160 by yourself.

If line 200 does in fact exist, BURP will properly renumber the reference to it in line 160. However, there's another problem that you've got to look out for. You've written this GOTO 200+X on the assumption that the arithmetic expression 200+X will yield the result 201, 202, or 203. If you now renumber your program using an increment value other than 1, you've got a problem. Although the renumbering may result in one or two of these values (201, 202, or 203) being a valid line number, unless you use an increment of 1, at least one of them will not exist.

Troubleshooting and Cautions

(Continued.....)

One final caution: I haven't checked this out, but I see the possibility of a problem if you have really filled a logical line to its maximum, and increase the size of that line's number by one or more digit positions.

For example, if line number 10 contains the maximum number of characters allowed on a logical BASIC line and, as the result of renumbering, it becomes line 100, you may have a problem.

I think that as long as your program remains in tokenized form, either in RAM or SAVEd on cassette or diskette, it will RUN without objection. If you LIST your program on cassette or diskette, the correct un-tokenized statements will likely be written to the device. However, you might have a problem when you subsequently attempt to ENTER that LISTed program into RAM. Why? Because, in the example above, increasing the length, in digits, of the line number may have shoved the last characters of your BASIC line beyond the maximum allowable logical line length. When BASIC is reading your ENTERed statements, it may object to this.

V. ADVANCED TECHNICAL INFORMATION

Two versions of BURP are on the diskette and one is on the cassette. One version is condensed ("mashed") to achieve maximum performance and minimum RAM space requirement. This version should be considered your operational version. It's on both diskette and cassette.

The second version on the diskette is uncondensed, containing REMarks. It is intended to provide you with a readable program that you can follow with a minimum of difficulty, to satisfy your curiosity about how the program works. While reading the following description of the functions performed by various sections of the program, refer to the uncondensed version of BURP to see the corresponding BASIC statements. You can either display these lines on your screen (if you have the program on diskette and entered into RAM) or look over the program listing in Section VI.

Program Narrative

Lines 32500 - 32518:

Before BURP begins executing, it issues a CLR statement to insure that all free RAM is made available. It then clears the screen, and requests the new starting line number and the increment you wish to use. Note that if a typing error is made in INPUTting either of these values, the error will TRAP to a re-prompt for both of these variables.

Lines 32519 - 32525:

Poking a "1" into location 752 suppresses display of the cursor. Then BURP DIMensions TL (an array used in tokenizing line numbers) and YORN$ (a string variable used to accept responses).

Lines 32526 - 32535:

Initializes MAXTST (used to insure that new line numbers do not exceed 32499) equal to the new starting line number (NEWNR). BURP then calculates the capacity (CAP) of the line number/reference address table, using all available free RAM. The two arrays (OLDNR and RAMADD) comprising the table are then DIMensioned by CAP.

Program Narrative

(Continued....)

Lines 32536 - 32542:

Sets the screen position to display the current line numbers as the corresponding BASIC lines are scanned. Obtains the beginning RAM address (P) of the BASIC program by PEEKing at locations 136 and 137.

Lines 32543 - 32544:

Gets the line number (LNMBR) of the current line by PEEKing at the first two bytes of the line. Then gets the length of the line (LNTH) by PEEKing at the third byte.

Lines 32545 - 32547:

When line number (LNMBR) 32500 is reached, terminate the scanning of the BASIC program and GOTO the next step, sorting the references and POKEing the new line numbers into RAM.

Lines 32548 - 32555:

Increment the maximum line number control variable (MAXTST) by the increment value (INCREMENT). If it exceeds 32499, notify the operator that the maximum allowable line number has been reached, and terminate the program.

Lines 32556 - 32560:

Set the screen position and display the number of the BASIC line that is currently being scanned. Execute the subroutine at line 32566 to scan the line for instructions that might reference line numbers. On return from the subroutine, increment the pointer (P) to the beginning of the current line of BASIC by adding the line length (LNTH) to it. Then repeat the process, scanning the next successive line.

Lines 32561 - 32571:

Get the current instruction address (INSADD) from the 4th byte of the line currently being scanned. Get the length of the current line (LL) from the 3rd byte of the line currently being scanned. Using the instruction address (INSADD), get the length (IL) of the instruction about to be scanned. From the 2nd byte of the instruction, get the tokenized instruction code (OP) and the byte following it (F).

Program Narrative

(Continued....)

Lines 32572 - 32581:

Examine the tokenized instruction code (OP) and the byte following it (F) to determine if the combination indicates that it is an instruction that could contain a reference to a line number. If it is, execute one of the subroutines to examine the operands of the instruction to see if they are explicit line numbers or variables.

Lines 32582 - 32590:

If the tokenized instruction code (OP) was a "7", this is an IF statement. This subroutine will scan the IF statement to see if it contains any implied or explicit GOTOs.

Lines 32591 - 32605:

If the tokenized instruction code (OP) was a "30" or a "4", this is an ON or LIST statement. Scan it for line number operands.

Lines 32606 - 32619:

This is the beginning of the subroutine that enters the referenced line numbers which were found embedded in the BASIC statement, together with the RAM address of each reference, into the table of references (OLDNR/RAMADD). Upon entry, REFADD points to the address within the BASIC line of the tokenized reference. Lines 32616 to 32619 break the tokenized line number down into 4 discrete bytes (BY1, BY2, BY3, and BY4).

Line 32620:

Once the line number reference address has been established, execute the subroutine at line 32649, which will attempt to convert the tokenized line number to a decimal value.

Lines 32621 - 32638:

On returning from the tokenized to decimal conversion subroutine, the variable OPRND should contain the decimal value of the referenced line number. However, if the result of the conversion attempt (OPRND) is equal to zero, this indicates that a variable was encountered. In this case, the

Program Narrative

(Continued....)

line containing the variable reference is displayed on the screen and the operator asked if he would like to continue, or terminate the program. The cursor is restored by POKE-ing 0 into location 752, and the operator is prompted for a reply (YORN$). The only valid response for continuation is an upper-case "Y". If the operator elects to continue (YORN$="Y"), the displayed line and the prompt and its response are erased from the screen by 8 repetitions of PRINT "Shift/Delete", and the program continues.

Lines 32639 - 32648:

If the tokenized to decimal conversion subroutine returned a valid decimalized line number (OPRND), increment the variable REFCNT, which indicates the number of line number references detected so far. This is compared against our calculated capacity (CAP), which depended on the amount of free RAM available. If REFCNT exceeds our capacity, the operator is notified and the process is terminated. Otherwise, the line number referenced, together with the actual RAM address of its tokenized occurrence, are added to the table of references (OLDNR/RAMADD).

Lines 32649 - 32661:

These lines constitute the subroutine that does the conversion of a tokenized line number to a decimal value in variable OPRND.

Lines 32662 - 32682:

This is a subroutine that converts a decimal line number to its tokenized form, the result being a four element array TL.

Lines 32683 - 32696:

Upon completing the scan of the BASIC program, having loaded the table of references and their addresses, we come to line 32692. At this point, execute the subroutine at 32725, which will sort the table by line number, to expedite subsequent searching of the table. SRST is initialized to 1, to cause our searching to begin, initially, at the beginning of the table. SRST will be incremented each time we find a line number referenced in the table, thereby decreasing the

Program Narrative

(Continued....)

time required for subsequent searches of the table. Set the screen position, display an indication that we have entered the actual renumbering process, and set P to once again point to the beginning of the BASIC program.

Lines 32697 - 32713:

Once more, obtain the original line number (LNMBR) from the first 2 bytes of the line, and the length of the line (LL) from the 3rd byte. Using the new starting line number (NEWNR) as our starting value, develop a new line number. Then proceed to search the table, to see if the current line number was referenced anywhere. If it was, execute the subroutine at line 32753, which will tokenize the new line number (NEWNR) and POKE it into the RAM address which was obtained from the table. Then POKE the new line number into the first 2 bytes of the line itself, and then repeat the process for the next line of the BASIC program.

Lines 32714 - 32720:

Upon successful completion of the renumbering, the cursor display is once again enabled and a tone generated to signal completion. A CLR is used to purge RAM of the no longer needed arrays, and the BURP ENDs.

Lines 32721 - 32747:

This is the subroutine that sorts the table of line number references and their corresponding RAM addresses. It is a simple version of a "bubble" sort. The number of references is displayed, followed by a blinking asterisk, for the purpose of reassuring you that something productive is happening, and to give some indication of BURP's progress.

Lines 32749 - 32761:

This subroutine POKEs the new line numbers into the RAM locations where they are referenced. Before doing so, however, they must be tokenized by executing the subroutine at line 32666. On return from this subroutine, the four element array TL contains the tokenized equivalent of the new line number.

BURP Variables

The variable names used in the unmashed version of BURP were chosen to be somewhat descriptive of their function. However, in order to minimize the time required to ENTER the BURP program in its operational or "mashed" form, short, non-descript variable names were substituted. The reason for this should be apparent when you consider that, in order to merge it with your BASIC program in RAM, using the ENTER statement, BURP must be maintained in untokenized form on a tape cassette or diskette. The shorter the variable names, the fewer bytes must be read from the cassette or diskette. Additionally, more statements could be "mashed" into one line if the variable names are short.

Consequently, the "mashed" version of BURP, which is the one you should use, was translated to abbreviated variable names. The following "dictionaries" are presented to give not only an indication of the use of each variable, but also to provide a cross-reference of the variable names used in each version. You will notice that, in the mashed version, some variables are used for several different purposes, in order to conserve variable names.

Variables used in BURP

(Sequenced by Unmashed Variable Name)

Mashed	Unmashed	Use
B2-4	BI2-4	Integer value of BY2-4/16.
Y1-4	BY1-4	Breakdown of 4 elements of a tokenized line number.
G	CAP	Capacity of the OLDNR/RAMADD tables.
E	D	FOR loop control in sorting routine.
F	F	The byte following the instruction code in a line. Also used as a temporary holding area for RAMADD during sorting.
R	IL	Length of the instruction currently being scanned.
B	INCREMENT	New line number increment value.
N	INSADD	Address of instruction currently being scanned.
X	J	Controls FOR loops.
U	LH	Used as work area in tokenizing decimal line numbers.
Q	LL	Line length.
L	LNMBR	Number of line currently being operated upon.
M	LNTH	Length of line currently being operated upon.
V	LT	Used as work area in tokenizing decimal line numbers.
E	MAXTST	Variable used to control protection against new line numbers greater than 32499.
A	NEWNR	New starting line number, subsequently incremented.
H	OLDNR	Table (array) of old line numbers referenced.

Variables used in BURP

(Continued.....)

(Sequenced by Unmashed Variable Name)

Mashed	Unmashed	Use
S	OP	Tokenized instruction code of the instruction currently being scanned.
Z	OPRND	Referenced line number converted to decimal value.
P	P	Current position in RAM being operated upon.
T	PTR	Controls FOR loops in scanning lines.
K	RAMADD	Table (array) of RAM addresses of line number references.
U	REFADD	Address of a line number reference within a line.
AA	REFCNT	Number of line number references detected.
SR	SRST	Points to position in the table to begin a search.
C	TL	Used to create 4-element tokenized line numbers.
D$	YORN$	Used to INPUT response to yes/no queries.

Variables used in BURP

(Sequenced by Mashed Variable Name)

Mashed	Unmashed	Use
A	NEWNR	New starting line number, subsequently incremented.
AA	REFCNT	Number of line number references detected.
B	INCREMENT	New line number increment value.
B2-4	BI2-4	Integer value of BY2-4/16.
C	TL	Used to create 4-element tokenized line numbers.
D$	YORN$	Used to INPUT response to yes/no queries.
E	D	FOR loop control in sorting routine.
E	MAXTST	Variable used to control protection against new line numbers greater than 32499.
F	F	The byte following the instruction code in a line. Also used as a temporary holding area for RAMADD during sorting.
G	CAP	Capacity of the OLDNR/RAMADD tables.
H	OLDNR	Table (array) of old line numbers referenced.
K	RAMADD	Table (array) of RAM addresses of line number references.
L	LNMBR	Number of line currently being operated upon.
M	LNTH	Length of line currently being operated upon.

Variables used in BURP

(Continued.....)

(Sequenced by Mashed Variable Name)

Mashed	Unmashed	Use
N	INSADD	Address of instruction currently being scanned.
P	P	Current position in RAM being operated upon.
Q	LL	Line length.
R	IL	Length of the instruction currently being scanned.
S	OP	Tokenized instruction code of the instruction currently being scanned.
SR	SRST	Points to position in the table to begin a search.
T	PTR	Controls FOR loops in scanning lines.
U	LH	Used as work area in tokenizing decimal line numbers.
U	REFADD	Address of a line number reference within a line.
V	LT	Used as work area in tokenizing decimal line numbers.
X	J	Controls FOR loops.
Y1-4	BY1-4	Breakdown of 4 elements of a tokenized line number.
Z	OPRND	Referenced line number converted to decimal value.

BURP Constants

Numeric constants are values expressed by a number (1,024 for instance) rather than represented by a variable name (OLDNR for instance)

The unmashed version of BURP uses numeric constants in some of the calculations because they are "constant" values, and their numeric representation more clearly indicates the nature of the calculation being performed.

However, in a tokenized BASIC statement, constants occupy more bytes than a variable reference. Therefore, in order to reduce the RAM requirement of the "mashed" version of BURP, some of these constant values were replaced by variables. The use of these variables, initialized to the required constant value, reduce the size of BURP.

The variables and their equivalent numeric constant values are as follows:

Constant Value	Variable Equivalent Used in Mashed BURP
1	W
2	W2
16	FF
10	FG
100	FH
1000	FI

Further Optimization of BURP

With the benefit of hindsight, the author sees several opportunities for improving BURP, some of which would increase its RAM requirement, and others which would have as their objective the reduction of its size.

When BURP realizes that it has insufficient RAM to hold all of the line number references, it informs you of that fact and quits. You might try to beat me to modifying that piece of the program so that, rather than abruptly quitting, BURP will continue to scan the program to its end. Having set a switch when the capacity limit was detected, it could then terminate - but first display the total reference count as compared to the array capacity available. In this way, you would have some idea of the extent to which you exceeded the array capacity. If you missed by only a couple of bytes of RAM (each line number reference requires 12 bytes of array space), you could perhaps temporarily delete a REM statement or two from your BASIC program, do your renumber, and then replace the REM statements.

One of these days, when the author can afford an Atari 810 Disk Drive, he'd like to break BURP up into three or more segments. Each would perform one of the basic phases of the process and, upon concluding its task, bring the next phase segment into RAM on top of itself. This could significantly reduce the space requirements of BURP. Maybe you'd like to try it yourself.

BURP was written without access to any documentation whatsoever of the internals of the BASIC interpretation and tokenizing scheme. I discerned the tokenized instruction codes and line number representation by typing in a BASIC statement and then PEEKing at the tokenized line. If you can come up with a more efficient routine for doing the conversion from a decimal value to a tokenized line number, and vice versa, I'd like to see it, as my manipulations seem to me to be rather cumbersome.

You might like to make BURP a little more entertaining to run by zipping it up with various pleasant (please, nothing obnoxious) sounds that would give you an indication of its progress without having to look at the screen.

Further Optimization of BURP

(Continued.....)

The foregoing was primarily for those of you who've got RAM to spare. A deliberate effort was made, however, to provide a renumbering utility that was usable by those people who haven't got a full house. If you're among that group, you'll likely be looking to further "mash" BURP, sacrificing cosmetic effects for raw function.

For those of you who would like to squeeze BURP further, you might want to consider:

- Removing the display of information that lets you know what BURP is doing.

- Remove the TRAP on INPUT at the beginning of the program. If the user INPUTs a non-numeric value when you're prompted for the new starting line number or increment, you could just let BURP abort itself. No harm will be done.

- Remove the test for new line numbers greater than 32499. You would then have to rely on your own mental arithmetic to insure that you don't violate this rule. Otherwise BURP will, in effect, commit suicide by attempting to renumber itself.

- In preparing these notes, I've noticed that I may have overlooked opportunities to use a variable for multiple purposes. You may want to contemplate doing this for the few bytes it will yield.

Finally, if you really want to make BURP go as fast as possible, and don't care to see what's happening, you can try disabling Direct Memory Access (DMA) by POKEing zero into location 559. There were a couple of magazine articles published which pointed out that this could yield significant processor performance improvements. I won't go into detail on this. However, you should know that the 6502 processor has cycles "stolen" from it by one of the outboard chips for the simple task of maintaining the image on your TV screen. Suppressing this chip's access to RAM suppresses the TV image, as well as any other I/O activity (including keyboard), but lets the 6502 dedicate its full resource to the program its running.

Further Optimization of BURP

(Continued.....)

I have not tried it with BURP, but you could disable DMA during the lengthy processes - scanning the lines, sorting the table, and POKEing the new numbers in - but you would have to enable it, by POKEing 34 into location 559, every time it was necessary to advise the user of an anomalous occurrence, and prompt for his response.

The Secret Numbers

As I said earlier, I PEEKed around at a lot of tokenized BASIC statements in order to decipher their format and the tokenized representations for various operations.

In some cases, a different decimal token is used to represent the same operation, depending on whether the operation is the first on a line of BASIC statements, or is embedded within the line.

For example, in the tokenized line:

```
235 GOTO 420
```

the GOTO would be tokenized as a decimal 10.

On the other hand, in the tokenized line:

```
235 LET A=B:GOTO 420
```

the GOTO would be tokenized as a decimal 23.

Listed below are the BASIC statements that can or must reference line numbers, and the decimal value of the byte that BASIC uses to represent their tokenized equivalents.

Operation	Token	
LIST	4	
GOTO	10	(23 when embedded)
GO TO	11	
GOSUB	12	(24 when embedded)
TRAP	13	
RESTORE	35	
THEN	27	(occurs as implied GOTO)
ON	30	

The Secret Numbers

(Continued...)

The process of identifying these tokenized operations is not, however, all that simple. For example, a decimal 4 is also used to represent an OPEN. The LIST operation may or may not reference line numbers and may have a device code operand. References to line numbers are optional in the RESTORE operation. IF statements, tokenized as a decimal 7, must be scanned for the possibility of implied as well as explicit GOTO's.

References to variables are tokenized as a decimal value equal to the number of the variable with respect to its position in the variable table + 128.

I will make no attempt at explaining the scheme of tokenizing numeric constants and references to line numbers. While it seems terribly wasteful of space, the authors of Atari BASIC undoubtedly did it this way for good reason. In any event, it requires 7 bytes of memory, even if the value represented is 1.

Here are some examples of what a tokenized numeric constant will look like:

50 = 14,64,80,0,0,0,0

504 = 14,65,5,4,0,0,0

1800 = 14,65,24,0,0,0,0

9999 = 14,65,153,153,0,0,0

32012 = 14,66,3,32,18,0,0

In each case, the 14 indicates to the BASIC interpreter that a constant follows. The examples I have shown all fall within the range of allowable line numbers. Numeric constants can, however, exceed the maximum value of 32767 permitted for line numbers. But, in every case where the tokenized value represents a line number, the last two bytes are always a null (0) value.

I hope you find BURP to be worthwhile, not only for the function it provides, but as an interesting excercise in increasing your understanding, appreciation, and enthusiasm for personal computing.

If you have any problems with BURP, or discover better methods of performing some of the functions, I invite you to write to me at the address below.

Happy computing!

Gifford O. Kucsma
R. D. 1, Prescott Circle
Lebanon, New Jersey
08833

```
32500 REM ---B.U.R.P.-----(Rel. 1.1)-
32501 REM
32502 REM BASIC Utility
32503 REM for Renumbering Programs
32504 REM
32505 REM -------by G. O. Kucsma-----
32506 REM Clear all free RAM. Then
32507 REM get the new starting line
32508 REM number and the increment.
32509 CLR
32510 PRINT "}"
32511 PRINT "New start line number";
32512 TRAP 32510
32513 INPUT NEWNR
32514 PRINT
32515 PRINT "Increment";
32516 INPUT INCREMENT
32517 TRAP 40000
32518 PRINT
32519 REM --------------------------
32520 REM Suppress the cursor,
32521 REM DImension the variables.
32522 REM
32523 POKE 752,1
32524 DIM TL(4)
32525 DIM YORN$(1)
32526 REM --------------------------
32527 REM Initialize control of maxi-
32528 REM mum allowable line #, then
32529 REM calculate the available RAM
32530 REM and dimension the reference
32531 REM number table to use it all.
32532 REM
32533 LET MAXTST=NEWNR
32534 LET CAP=INT((FRE(0)-90)/12)
32535 DIM OLDNR(CAP),RAMADD(CAP)
32536 REM --------------------------
32537 REM Main Loop. Scan the pro-
32538 REM gram, line by line.
32539 REM
32540 POSITION 2,5
32541 PRINT "Scanning line:";
32542 LET P=PEEK(136)+PEEK(137)*256
32543 LET LNMBR=PEEK(P)+PEEK(P+1)*256
32544 LET LNTH=PEEK(P+2)
32545 IF LNMBR<32500 THEN GOTO 32548
32546 PRINT
32547 GOTO 32692
32548 LET MAXTST=MAXTST+INCREMENT
32549 IF MAXTST<32500 THEN GOTO 32556
32550 PRINT
32551 PRINT
32552 PRINT "New line number exceeds"
32553 PRINT "the maximum allowable"
32554 PRINT "value.}"
32555 POKE 752,0:GOTO 32631
32556 POSITION 17,5
32557 PRINT LNMBR;
32558 GOSUB 32566
32559 LET P=P+LNTH
32560 GOTO 32543
32561 REM --------------------------
```

```
32562 REM Subroutine to scan line for
32563 REM operations with line number
32564 REM operands.
32565 REM
32566 LET INSADD=P+3
32567 LET CUMLENTH=0
32568 LET LL=PEEK(P+2)
32569 LET IL=PEEK(INSADD)
32570 LET OP=PEEK(INSADD+1)
32571 LET F=PEEK(INSADD+2)
32572 IF ((OP=4) AND ((F<>20) AND (F<>22))) THEN GOSUB 32595:GOTO 32579
32573 IF OP=7 THEN GOSUB 32585:GOTO 32579
32574 IF OP=30 THEN GOSUB 32595:GOTO 32579
32575 IF (OP>9) AND (OP<14) THEN GOSUB 32614:GOTO 32579
32576 IF OP=23 AND F=28 THEN 32579
32577 IF (OP=23) OR (OP=24) OR (OP=27) THEN GOSUB 32614:GOTO 32579
32578 IF (OP=35) AND (F<>22) THEN GOSUB 32614
32579 LET INSADD=P+IL
32580 IF INSADD=P+LL THEN RETURN
32581 GOTO 32569
32582 REM ----------------------------
32583 REM Scan 'IF' statements.
32584 REM
32585 FOR PTR=INSADD+3 TO (P+IL)-1
32586 IF PEEK(PTR)=14 THEN GOTO 32589
32587 NEXT PTR
32588 RETURN
32589 IF (PEEK(PTR-1)=10) OR (PEEK(PTR-1)=27) THEN REFADD=PTR+1:GOSUB 32616
32590 GOTO 32587
32591 REM ----------------------------
32592 REM Scan 'ON' and 'LIST' state-
32593 REM ments.
32594 REM
32595 LET SP=3
32596 IF OP=4 THEN LET SP=1
32597 FOR PTR=INSADD+SP TO (P+IL)-1
32598 IF (PEEK(PTR)=4 OR PEEK(PTR)=23 OR PEEK(PTR)=24 OR PEEK(PTR)=18) AND PEEK(
PTR+1)=14 THEN GOTO 32602
32599 IF (PEEK(PTR)=4 OR PEEK(PTR)=23 OR PEEK(PTR)=24 OR PEEK(PTR)=18) AND PEEK(
PTR+1)>127 THEN GOSUB 32622
32600 NEXT PTR
32601 RETURN
32602 LET REFADD=PTR+2
32603 GOSUB 32616
32604 LET PTR=PTR+7
32605 GOTO 32600
32606 REM ----------------------------
32607 REM Set pointers to operand,
32608 REM go convert to decimal,
32609 REM and if not a line number,
32610 REM show the line, pause, and
32611 REM give operator option of
32612 REM terminating the program.
32613 REM
32614 LET REFADD=INSADD+3
32615 REM
32616 LET BY1=PEEK(REFADD)
32617 LET BY2=PEEK(REFADD+1)
32618 LET BY3=PEEK(REFADD+2)
32619 LET BY4=PEEK(REFADD+3)
32620 GOSUB 32649
32621 IF OPRND<>0 THEN GOTO 32639
32622 PRINT
32623 LIST LNMBR
32624 PRINT
32625 PRINT "Variable reference.>"
```

```
32626 PRINT
32627 PRINT "Go ahead anyhow";
32628 POKE 752,0
32629 INPUT YORN$
32630 IF YORN$(1,1)="Y" THEN 32633
32631 CLR
32632 END
32633 POKE 752,1
32634 POSITION 2,7
32635 FOR X=1 TO 8
32636 PRINT "";
32637 NEXT X
32638 GOTO 32648
32639 LET REFCNT=REFCNT+1
32640 IF REFCNT<=CAP THEN GOTO 32646
32641 PRINT
32642 PRINT "Insufficient RAM to"
32643 PRINT "handle more than ";CAP
32644 PRINT "references.>"
32645 POKE 752,0:GOTO 32631
32646 LET RAMADD(REFCNT)=REFADD
32647 LET OLDNR(REFCNT)=OPRND
32648 RETURN
32649 REM --------------------------
32650 REM Subroutine to convert
32651 REM tokenized line number to
32652 REM decimal.
32653 REM
32654 LET BI2=INT(BY2/16)
32655 LET BI3=INT(BY3/16)
32656 LET BI4=INT(BY4/16)
32657 LET OPRND=0
32658 IF BY1=64 THEN OPRND=(BI2*10)+(BY2-(BI2*16))
32659 IF BY1=65 THEN OPRND=(BI2*1000)+((BY2-(BI2*16))*100)+(BI3*10)+(BY3-(BI3*16
))
32660 IF BY1=66 THEN OPRND=(BY2*10000)+(BI3*1000)+((BY3-(BI3*16))*100)+(BI4*10)+
(BY4-(BI4)*16)
32661 RETURN
32662 REM --------------------------
32663 REM Subroutine to tokenize a
32664 REM decimal line number.
32665 REM
32666 FOR J=1 TO 4
32667 LET TL(J)=0
32668 NEXT J
32669 IF NEWNR>99 THEN GOTO 32672
32670 LET TL(1)=64
32671 LET TL(2)=((INT(NEWNR/10))*16)+(NEWNR-(INT(NEWNR/10)*10)):RETURN
32672 IF NEWNR>9999 THEN GOTO 32677
32673 LET TL(1)=65
32674 LET TL(2)=(INT(NEWNR/1000)*16)+INT((NEWNR-((INT(NEWNR/1000))*1000))/100)
32675 LET LH=NEWNR-INT(INT(NEWNR/100)*100)
32676 LET TL(3)=(INT(LH/10)*16)+INT(LH-(INT(LH/10))*10):RETURN
32677 LET TL(1)=66
32678 LET TL(2)=INT(NEWNR/10000)
32679 LET LT=NEWNR-INT(INT(NEWNR/10000)*10000)
32680 LET TL(3)=(INT(LT/1000)*16)+INT((LT-((INT(LT/1000))*1000))/100)
32681 LET LH=LT-INT(INT(LT/100)*100)
32682 LET TL(4)=(INT(LH/10)*16)+INT(LH-(INT(LH/10))*10):RETURN
32683 REM --------------------------
32684 REM Go sort the table of refer-
32685 REM ences, then renumber the
32686 REM lines, looking up old lines
32687 REM in the table and poking new
32688 REM tokenized line number into
32689 REM RAM locations where
```

```
32690 REM referenced.
32691 REM
32692 GOSUB 32725
32693 LET SRST=1
32694 POSITION 2,9
32695 PRINT "Renumbering ";
32696 LET P=PEEK(136)+PEEK(137)*256
32697 LET LNMBR=PEEK(P)+PEEK(P+1)*256
32698 LET LL=PEEK(P+2)
32699 IF LNMBR>32499 THEN GOTO 32714
32700 POSITION 14,9
32701 PRINT LNMBR;" as ";NEWNR
32702 IF SRST>REFCNT THEN GOTO 32709
32703 FOR I=SRST TO REFCNT
32704 IF LNMBR<>OLDNR(I) THEN 32707
32705 GOSUB 32753
32706 GOTO 32708
32707 IF LNMBR<OLDNR(I) THEN 32709
32708 NEXT I
32709 POKE P+1,INT(NEWNR/256)
32710 POKE P,NEWNR-((INT(NEWNR/256))*256)
32711 LET NEWNR=NEWNR+INCREMENT
32712 LET P=P+LL
32713 GOTO 32697
32714 POKE 752,0
32715 SOUND 0,40,10,10
32716 FOR D=1 TO 30
32717 NEXT D
32718 SOUND 0,0,0,0
32719 CLR
32720 END
32721 REM ---------------------------
32722 REM Subroutine to sort the
32723 REM reference table.
32724 REM
32725 POKE 77,0
32727 PRINT
32728 PRINT "Sorting the references:"
32729 IF REFCNT=0 THEN 32746
32730 FOR D=1 TO REFCNT-1
32731 POSITION 26,7
32732 PRINT D;" ";
32733 IF OLDNR(D)<=OLDNR(D+1) THEN 32745
32734 FOR J=D TO 1 STEP -1
32735 PRINT "*";
32736 IF OLDNR(J)<=OLDNR(J+1) THEN 32745
32737 LET T=OLDNR(J)
32738 LET F=RAMADD(J)
32739 LET OLDNR(J)=OLDNR(J+1)
32740 LET RAMADD(J)=RAMADD(J+1)
32741 LET OLDNR(J+1)=T
32742 LET RAMADD(J+1)=F
32743 PRINT " ";
32744 NEXT J
32745 NEXT D
32746 RETURN
32747 REM
32748 RETURN
32749 REM ---------------------------
32750 REM Subroutine to poke new line
32751 REM numbers into RAM.
32752 REM
32753 GOSUB 32666
32754 FOR J=0 TO 3
32755 POKE RAMADD(I)+J,TL(J+1)
32756 NEXT J
```

```
32758 LET SRST=SRST+1
32759 RETURN
32760 REM
32761 REM ----------------------------
```

DISCLAIMER OF WARRANTY AND LIABILITY ON COMPUTER PROGRAMS

Neither Atari, Inc. ("ATARI"), nor its software supplier, distributor, or dealers make any express or implied warranty of any kind with respect to this computer software program and/or material, including, but not limited to warranties of merchantability and fitness for a particular purpose. This computer program software and/or material is distributed solely on an "as is" basis. The entire risk as to the quality and performance of such programs is with the purchaser. Purchaser accepts and uses this computer program software and/or material upon his/her own inspection of the computer software program and/or material, without reliance upon any representation or description concerning the computer program software and/or material. Should the computer program software and/or material prove defective, purchaser and not ATARI, its software supplier, distributor, or dealer, assumes the entire cost of all necessary servicing, repair, or correction, and any incidental damages.

In no event shall ATARI, or its software supplier, distributor, or dealer be liable or responsible to a purchaser, customer, or any other person or entity with respect to any liability, loss, incidental or consequential damage caused or alleged to be caused, directly or indirectly, by the computer program software and/or material, whether defective or otherwise, even if they have been advised of the possibility of such liability, loss, or damage.

LIMITED WARRANTIES ON MEDIA AND HARDWARE ACCESSORIES

ATARI warrants to the original consumer purchaser that the media on which the computer software program and/or material is recorded, including computer program cassettes or diskettes, and all hardware accessories are free from defects in materials or workmanship for a period of 30 days from the date of purchase. If a defect covered by this limited warranty is discovered during this 30-day warranty period, ATARI will repair or replace the media or hardware accessories, at ATARI's option, provided the media or hardware accessories and proof of date of purchase are delivered or mailed, postage prepaid, to the ATARI Program Exchange.

This warranty shall not apply if the media or hardware accessories (1) have been misused or show signs of excessive wear, (2) have been damaged by playback equipment or by being used with any products not supplied by ATARI, or (3) if the purchaser causes or permits the media or hardware accessories to be serviced or modified by anyone other than an authorized ATARI Service Center. Any applicable implied warranties on media or hardware accessories, including warranties of merchantability and fitness, are hereby limited to 30 days from the date of purchase. Consequential or incidental damages resulting from a breach of any applicable express or implied warranties on media or hardware accessories are hereby excluded. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. Some states also do not allow the exclusion or limitation of incidental or consequential damage, so the above limitation or exclusion may not apply to you.

ATARI PROGRAM EXCHANGE

REVIEW FORM

We're interested in your experiences with APX programs and documentation, both favorable and unfavorable. Many software authors are willing and eager to improve their programs if they know what users want. And, of course, we want to know about any bugs that slipped by us, so that the software author can fix them. We also want to know whether our documentation is meeting your needs. You are our best source for suggesting improvements! Please help us by taking a moment to fill in this review sheet. Fold the sheet in thirds and seal it so that the address on the bottom of the back becomes the envelope front. Thank you for helping us!

1. Name and APX number of program ______________________________

2. If you have problems using the program, please describe them here.

____

____

____

3. What do you especially like about this program?

____

____

____

4. What do you think the program's weaknesses are?

____

____

____

5. How can the catalog description be more accurate and/or comprehensive?

____

____

6. On a scale of 1 to 10, 1 being "poor" and 10 being "excellent", please rate the following aspects of this program?

_____ Easy to use
_____ User-oriented (e.g., menus, prompts, clear language)
_____ Enjoyable
_____ Self-instructuve
_____ Useful (non-game software)
_____ Imaginative graphics and sound

7. Describe any technical errors you found in the user instructions (please give page numbers).

8. What did you especially like about the user instructions?

9. What revisions or additions would improve these instructions?

10. On a scale of 1 to 10, 1 representing "poor" and 10 representing "excellent", how would you rate the user instructions and why?

11. Other comments about the software or user instructions:

```
 -----
|     |
|STAMP|
|     |
 -----
```

ATARI Program Exchange
P.O. Box 427
155 Moffett Park Drive, B-1
Sunnyvale, CA 94086

[seal here]